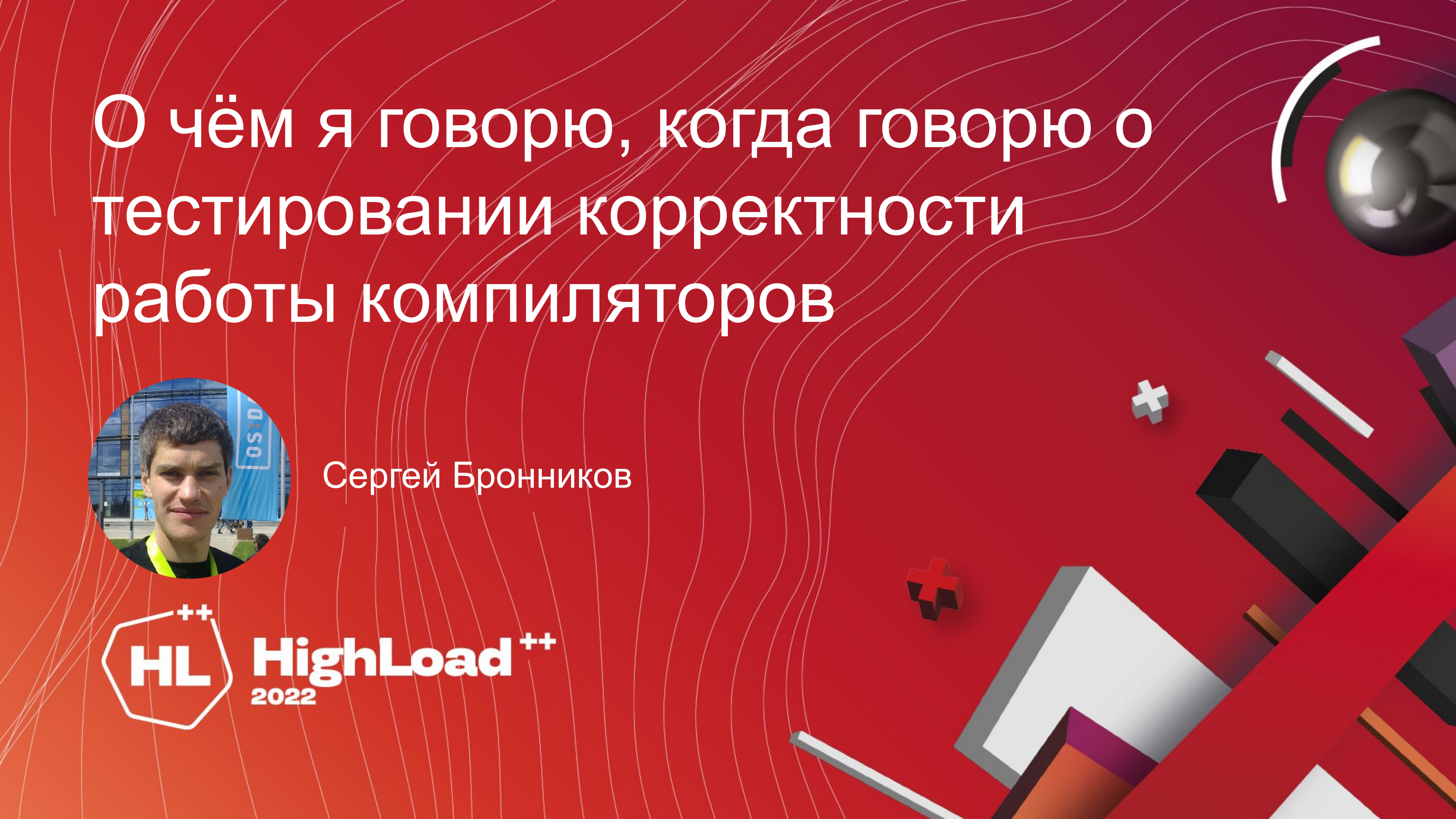


О чём я говорю, когда говорю о тестировании корректности работы компиляторов
Сергей Бронников
HL
HighLoad++
2022

План

- Введение
- Проблемы тестирования LuaJIT
- Подходы к решению
  - ★☆☆☆☆ Неструктурированный фаззинг
  - ★★☆☆☆ Синтаксический фаззинг
  - ★★★☆☆ Семантический фаззинг
  - ★☆☆☆☆ Сравнительный фаззинг
  - ★★★★☆ Тестирование оптимизаций
- Выводы

Введение

Tarantool

- Платформа in-memory-вычислений
- Описание бизнес-логики на Lua
- Используется собственный форк LuaJIT



LuaJIT



- Среда исполнения для Lua версии 5.1
- Трассирующий JIT компилятор
- Быстрее, чем референсная реализация:
  - x2-x4 без JIT–компиляции
  - x2-x100* с JIT–компиляцией
- Открытый код - лицензия MIT

Пользователи LuaJIT















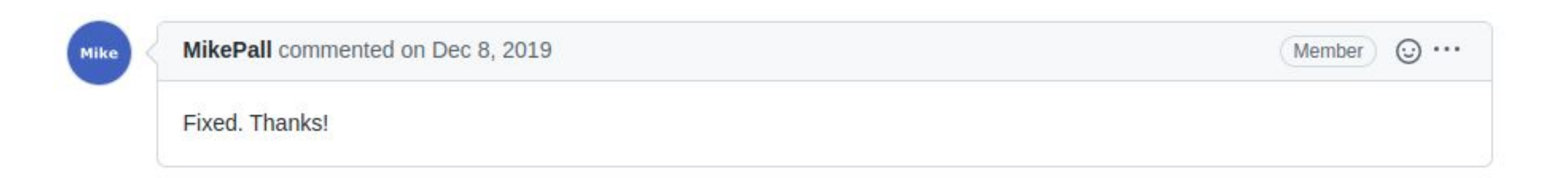
Проблемы

Проблемы тестирования LuaJIT

- Нет **собственного** набора тестов для компилятора
- Новые изменения от автора редко покрываются тестами
- В процессе эксплуатации Tarantool выявляются неприятные баги

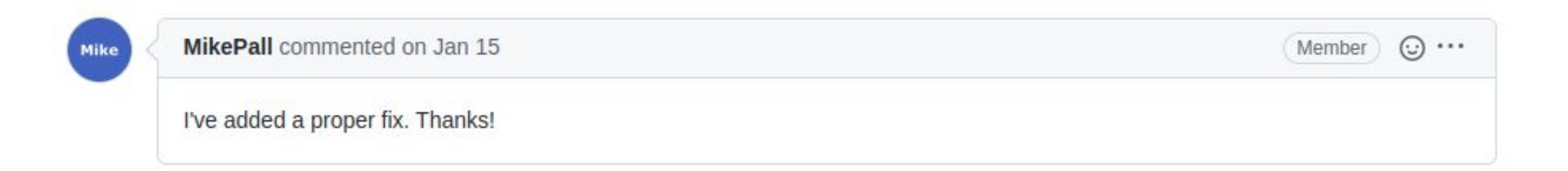
Пример бага: LuaJIT#528

```
> tonumber("-0.0")  -- -0
> tonumber("-0")    --  0
```



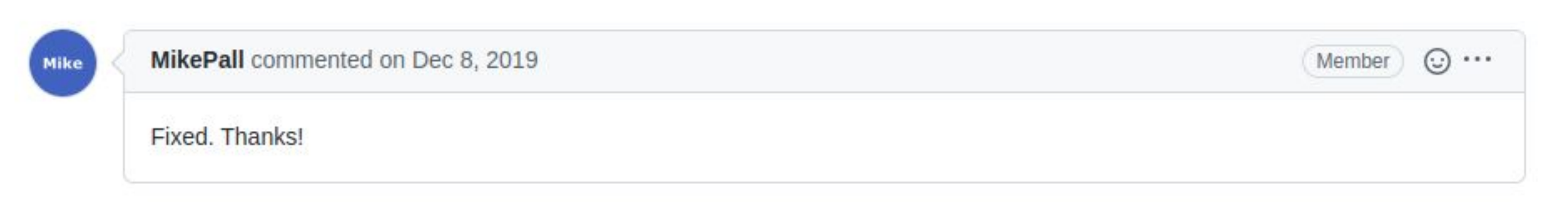
Пример бага: LuaJIT#528, спустя два года

```
> tonumber("-0")      -- 0
> tonumber("-0e1")    -- 0
> tonumber("-0x0")    -- 0
> tonumber("-0x0p1")  -- 0
```



Пример бага: LuaJIT#505

```
jit.opt.start("hotloop=1") -- включение агрессивной компиляции
for _ = 1, 20 do
  local value = "abc"
  local pos_c = string.find(value, "c", 1, true)
  local value2 = string.sub(value, 1, pos_c - 1)
  local pos_b = string.find(value2, "b", 2, true)
  assert(pos_b == 2, "FAIL: position of 'b' is " .. pos_b)
end
```



Тестирование LuaJIT в Tarantool

- Регрессионные тесты проекта PUC Rio Lua
- lua-Harness от Франсуа Перра (François Perrad)
- Собственные тесты для LuaJIT
- Запуск регрессионных тестов на каждый PR
- Покрытие кода тестами: 78% строк, 67% веток



Как мы работаем над стабильностью нашей реализации Lua

Антон Солдатов



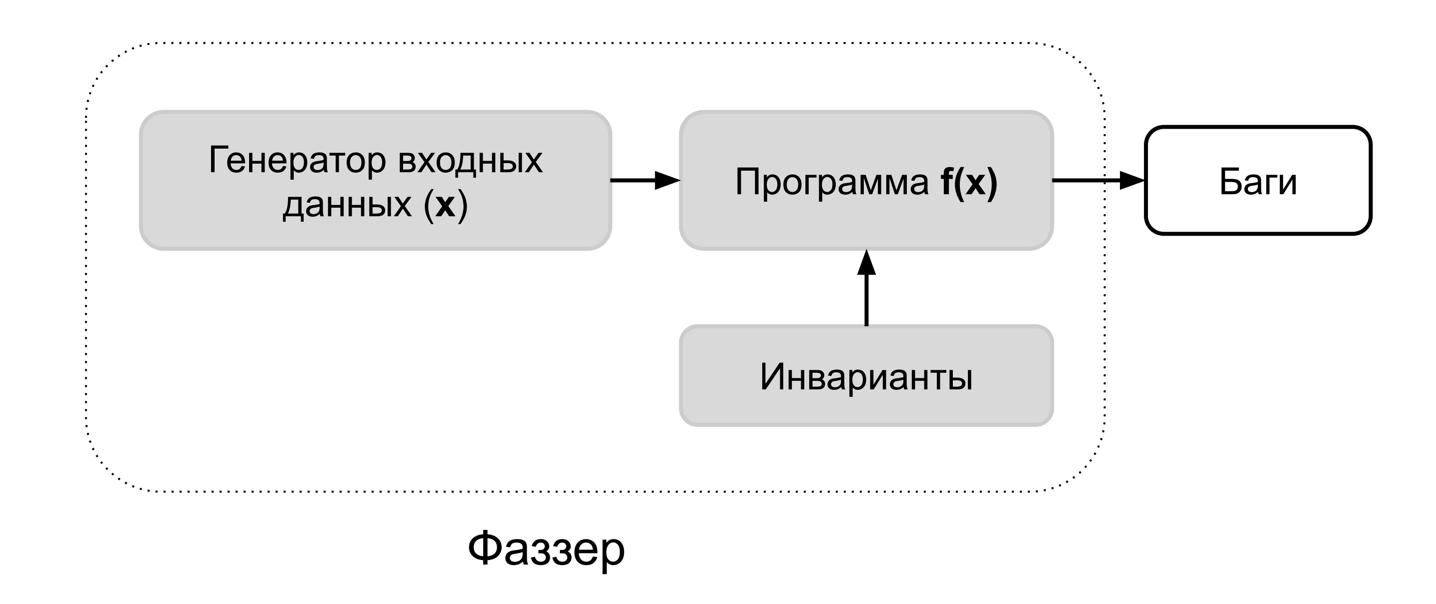


Тестирования всё ещё недостаточно!

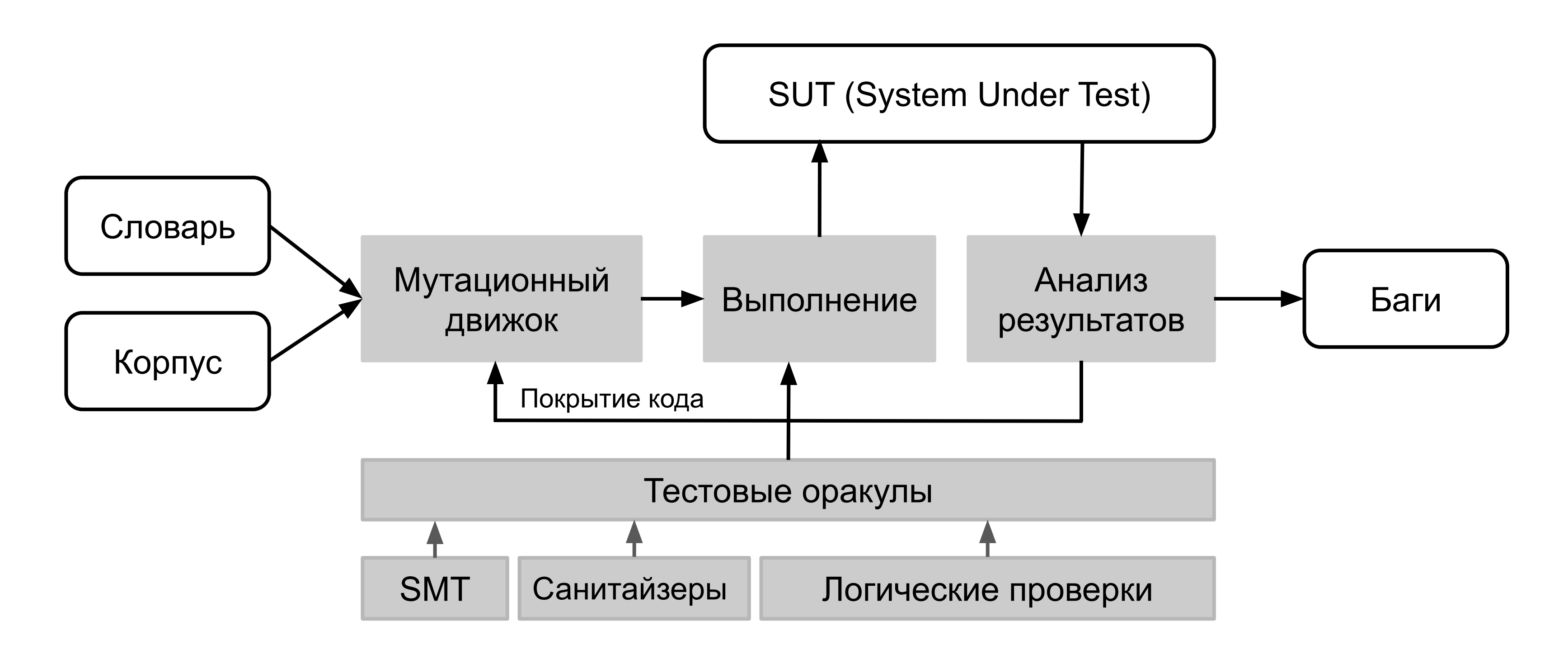
Принципиальные подходы к тестированию LuaJIT

- Тестирование с помощью примеров (example–based)
  - Тестирование “известного”: данные заранее известны
  - Arrange – Act – Assert
  - Все тесты для LuaJIT
- Рандомизированное тестирование
  - Тестирование “неизвестного”: данные всегда случайны
  - Идеально подходит для тестирования компиляторов
  - Есть научные работы, но без особенного успеха
  - Есть фаззер для PUC Rio Lua
  - Часть этого доклада

Упрощённая схема фаззинга



Детальная схема фаззинга



Результаты исследований

- “NAUTILUS: Fishing for Deep Bugs with Grammars”
  - **1** минорный баг в PUC Rio Lua
- “GRIMOIRE: Synthesizing Structure while Fuzzing”
  - **0** багов в PUC Rio Lua
- “Language-Agnostic Generation of Compilable Test Programs”
  - **0** багов в PUC Rio Lua

Подходы

> Неструктурированный фаззинг

Синтаксический фаззинг

Семантический фаззинг

Сравнительный фаззинг

Тестирование оптимизаций

Принцип

- Входные данные: текст из "корпуса" или случайные байты
- Мутации **на уровне байтов**: `InsertByte`, `EraseBytes` и т.д.
- Тестовый оракул
  - Санитайзеры: ASAN, UBSan и др.
  - Потребление памяти
  - Время выполнения

State of the art

- AFL++ – фаззинг фреймворк, форк AFL
- hongfuzz – фаззинг-фреймворк
- **libFuzzer** – C/C++ библиотека для фаззинга в LLVM

Реализация для LuaJIT

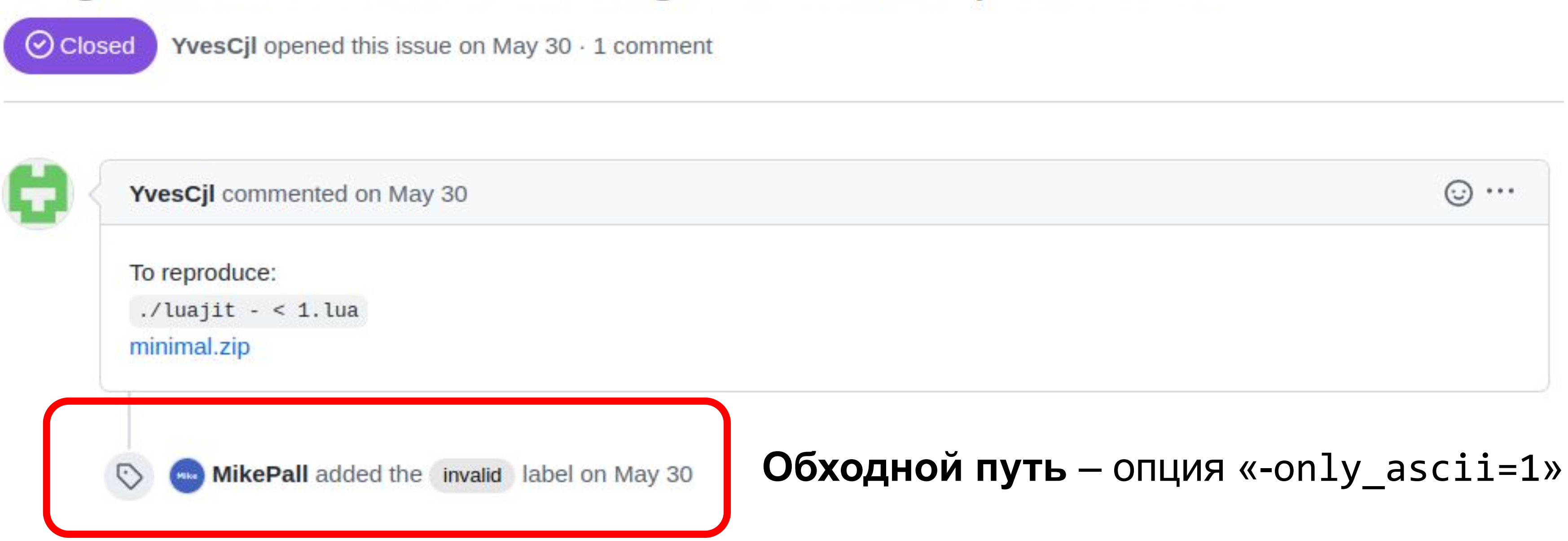
- Фаззер с помощью libFuzzer и Lua C API
- Корпус из **регрессионных тестов**
- Словарь на основе грамматики Lua 5.1
- Этот подход используется в фаззере для PUC Rio Lua (см. OSS Fuzz)

Неструктурированный фаззинг

```
$ hexdump reproducer.lua
0000000 001b
0000001
```

```
$ luajit reproducer.lua
luajit: lj_bcread.c:122: uint32_t bcread_byte(LexState *):
Assertion `ls->p < ls->pe' failed.
Aborted (core dumped)
$
```

“Неприемлемые” баги для LuaJIT



Обходной путь – опция «-only_ascii=1»

Пример случайной “программы”

```
print('testing strings and string library') assert('alo' < 'alo1') assfunctionert('' <
'a') assert('alo\0alo' < 'alo\0b') assert('alo\0alo\-\0' > 'alo\0alo\0') assert('alo' <
'alo\0') assert('alo\0' > 'a0\0') assert(not('\0=\p\0\0' < '\0\0\0')) assert('\0\0\0'
<= '\0\0\0\0') assert(not('\0\0\0\0' <= '\0\0\0')) assert('\0\0\0' <= '\0\0\0')
assert('\0\ "\foo) == n) assezt(print == nil  and 4) == "") local locales = { "ptb",
"ISO-8859-1", "pt_BR" } local function tryuuuuuuuuuuuuuuuuuuuuuuuuuuuuulocale (w) for
_, l in ipairs(localEs) do if os.setlocale(l& w) then return true end end return false
end if not trylo   le("collate")   then print("locale not supported") else <=
assert("alo" < "alo" and "alo" < "amo") end if not trylocale("ctypDe") then
print("locale not supported") else assert(string.gsub("aimsz", "%a", "debug.gethookI
"xxxxx") assert(string.gsub("aAiI", "%l", "x") == "xAxI") assert(string7.gsub("aAiI",
"%u", "x") == "axix") assert(string.up er"aAi{xuxu              "AAI{XUXU}GCO") end
os.setlocale("C") as

W Jsetlocale() == 'C') assert(os.setlocalu(nil, "numeric") == 'C') print('OK')
```

Pro et Contra

- ✓ Дешевизна и простота реализации
- ✓ Высокое покрытие кода при использовании корпуса и словаря
- ✓ Негативное тестирование синтаксического анализатора
- ✓ Сумасшедшая скорость – 261.000 запусков/сек.
- ✗ Нельзя использовать для проверки корректности работы компилятора

Подходы

Неструктурированный фаззинг

> **Синтаксический фаззинг**

Семантический фаззинг

Сравнительный фаззинг

Тестирование оптимизаций

Принцип

- Входные данные: текст из "корпуса" или случайное AST
- Мутации **по грамматике**
- Тестовый оракул
  - Санитайзеры (ASAN, UBSan и др.)
  - Потребление памяти
  - Время выполнения

Успешные реализации

- CSmith – фаззер для C/C++
  - “Finding and Understanding Bugs in C Compilers”
  - Найдено ~325 проблем
- YARPGen – фаззер для C/C++
  - “Random testing for C and C++ compilers with YARPGen”
  - Найдено ~220 проблем

Реализация для LuaJIT

- libFuzzer + Lua C API
- libProtobufMutator – библиотека для мутаций Protobuf
- Грамматика Lua 5.1 в формате Protobuf
- Сериализатор для структуры Protobuf

Пример случайной программы

Структура Lua-программы в Protobuf

```
chunk { stat { list { varlist { var { } } explist
{ explast { function { body { block { chunk { stat
{ list { varlist { var { } vars { indexname {
prefixexp { functioncall { prefArgs { prefixexp {
} args { tableconstructor { fieldlist { firstField
{ expression { binary { leftexp { str: "M" } binop
{ concat: 536870912 } rightexp { number:
2.1245307475328445 } } } } fields { field {
expression { binary { leftexp
"\001\000_\000\000\000\000\000" } bino
536870912 } rightexp { number:
-3.7319447282633243e-34 } } } } sep { } } } } } }
} } Name:
"qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqq" } } } explist {
explast { } } } } } } } } } } } semicolon: true }
stat { list { varlist { var { } vars { indexname {
prefixexp { functioncall { prefArgs { prefixexp {
} args { } } } } Name: "" } } vars { } } explist {
explast { number: 2.1210746765136719 } } } } }
```

Lua-программа

```lua
Name0 = function (  )
Name0, Name0 {'M' .. 2.124531,
' '            ..          -0.000000
qqqqqqqqqqqqqqqqqqqq
= 'Name'
end;
Name0, Name0 {  }.Name , Name0
= 2.121075
```

Нюансы сериализатора Protobuf

- Нужно учитывать семантику языка:
  - Не складывать строки: “attempt to perform arithmetic on a string value”
  - Не вычитать таблицы
  - Не вызывать несуществующие методы: “attempt to index a nil value”
  - …
- Избегать зацикливания в тестовых примерах

Pro et Contra

- ✓ Позитивное тестирование синтаксического анализатора
- ✓ Разнообразие синтаксически корректных программ
- ✗ Затратно по времени реализации (~1 месяц силами 1 студента)
- ✗ Простые мутаторы в libProtobufMutator

Подходы

Неструктурированный фаззинг

Синтаксический фаззинг

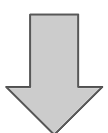
> Семантический фаззинг

Сравнительный фаззинг

Тестирование оптимизаций

Синтаксические мутации

меняем то, **как** код выглядит



Семантические мутации

меняем то, **что** код делает

Первая публикация по теме

Compiler validation via equivalence modulo inputs

Vu Le, Mehrdad Afshari, Zhendong Su

(2014, 384 ссылок)

Принцип

- Входные данные – случайные синтаксически-правильные программы
- Мутатор изменяет программу **без изменения семантики** (поведения)
- Тестовый оракул:
  - Санитайзеры: ASAN, UBSan и др.
  - Потребление памяти
  - Время выполнения
  - Динамические проверки из корпуса – `assert()`

Успешные реализации

- “FuzzIL: Coverage Guided Fuzzing for JavaScript Engines” – Samuel Groß
- “MongoDB's JavaScript fuzzer” – R. Guo
- JavaScript Raider

Реализация для LuaJIT

- libFuzzer позволяет интегрировать кастомный мутатор
- Интегрируем функцию `LLVMFuzzerCustomMutator()` с Lua
- Используем mulua для мутаций AST и сериализации обратно в Lua
- Реализуем мутации на Lua

Примеры мутаций

```
a = b and c    →  a = not not b and c

a = b + 1      →  a = (function() return b + 1 end)()

a = b + 1      →  for i = 1, 1 do a = b + 1 end

nil            →  collectgarbage()
```

Pro et Contra

- ✓ Большее разнообразие программ
- ✓ Фокусное тестирование
- ✓ Возможность использования логических проверок из корпуса
- ✗ Некоторые мутации компилятор может соптимизировать
- ✗ Низкая скорость – ~500 запусков/сек.

Подходы

Неструктурированный фаззинг

Синтаксический фаззинг

Семантический фаззинг

> **Сравнительный фаззинг**

Тестирование оптимизаций

Первая публикация по теме

Differential Testing for Software

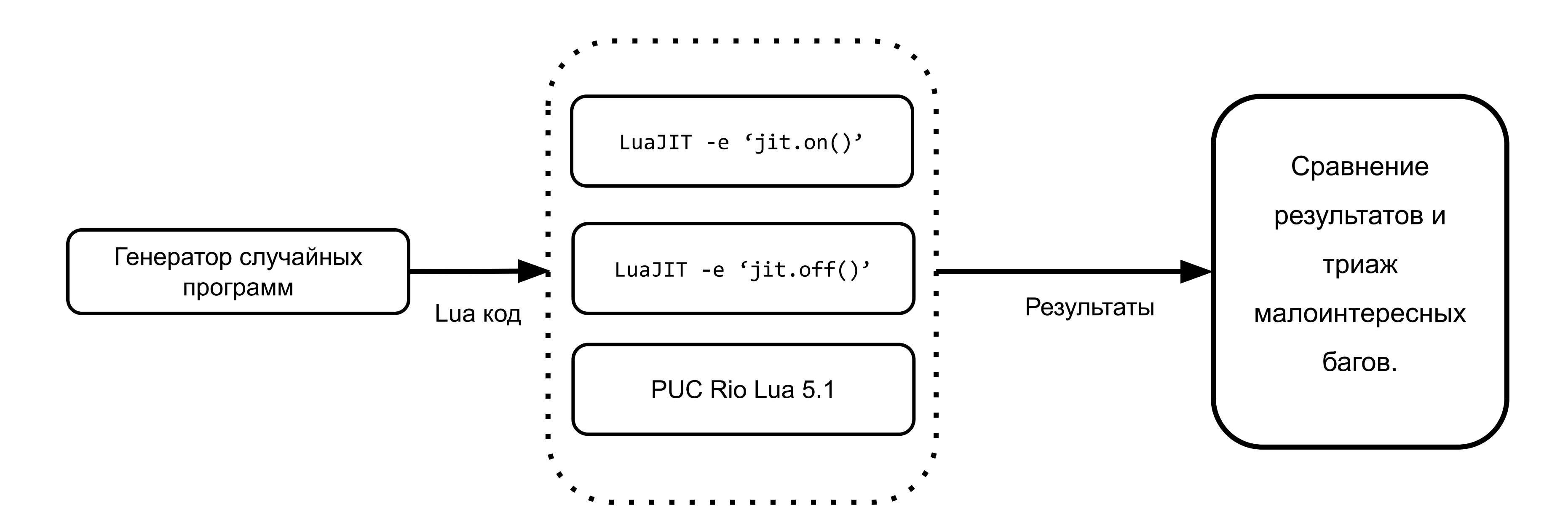
William M. McKeeman

(1998, 490 ссылок)

Принцип

- Генератор со случайными семантически-правильными программами
- Тестовый оракул – **более простая** или **другая** реализация

Принципиальная схема подхода



Успешные реализации

- Javascript: Jit-Picker, 32 бага
- Криптография: cryptofuzz, 160 багов
- DBMS: SQLancer, 450 багов
- Nezha, движок для сравнительного тестирования, 772 бага
- Машинное обучение: DeepXPlore

Реализация для LuaJIT

- libFuzzer + Lua C API
- Тестовые оракулы:
  - Референсная реализация - PUC Rio Lua
  - LuaJIT с выключенными оптимизациями
  - Санитайзеры: ASAN, UBSan и др.
  - Потребление памяти
  - Время выполнения

Pro et Contra

- ✓ Простота реализации теста
- ✓ Возможность найти проблемы в деталях реализации
- ✗ Не всегда применимо
- ✗ Ложноположительные срабатывания

Подходы

Неструктурированный фаззинг

Синтаксический фаззинг

Семантический фаззинг

Сравнительный фаззинг

> Тестирование оптимизаций

Принцип

- Входные данные – генератор с семантически-правильными программами
- Экспортируем BC/IR (промежуточное представление)
  - С включенными оптимизациями
  - Без оптимизаций
- Транслируем BC/IR в логические формулы
- Проверяем эквивалентность обеих формул с помощью SMT–решателя
- Тестовые оракулы:
  - Санитайзеры: ASAN, UBSan и др.
  - Время выполнения
  - Потребление памяти
  - SMT-решатель

Первая публикация по теме

TVOC: A Translation Validator for Optimizing Compilers

Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli & Lenore Zuck

(2005, 93 ссылки)

Используется в проектах

- LLVM: Alive/Alive2, ~70 багов
- GCC: pysmtgcc, 1 баг
- PyPy, 2 бага
- Язык описания контрактов Solidity
- SQL: Cosetta, 3 бага

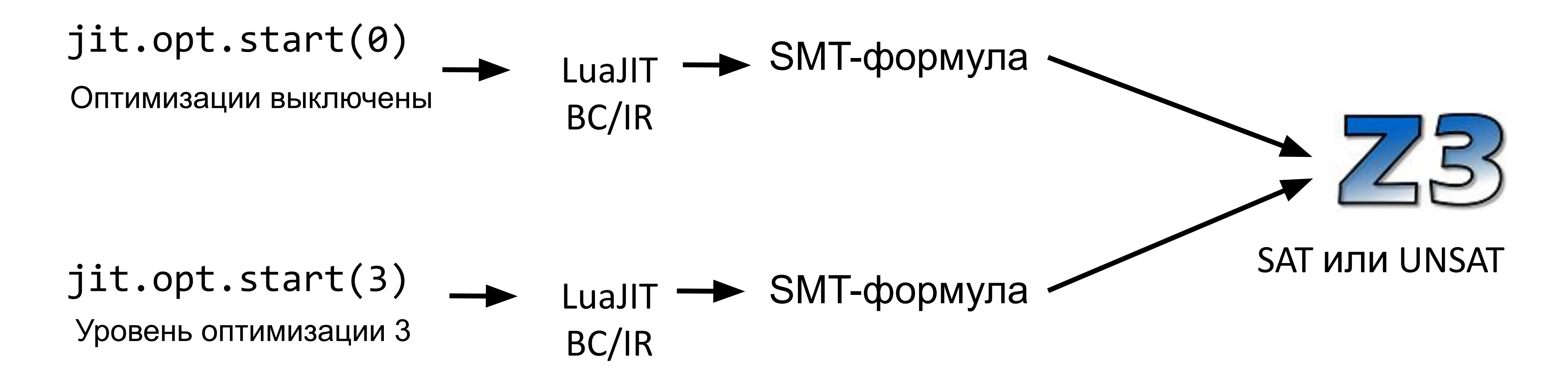
Пример бага от pysmtgcc: GCC#106523, не исправлено

```
unsigned char f7(unsigned char x, unsigned int y) {
    unsigned int t = x;
    return (t << y) | (t >> ((-y) & 7));
}
```

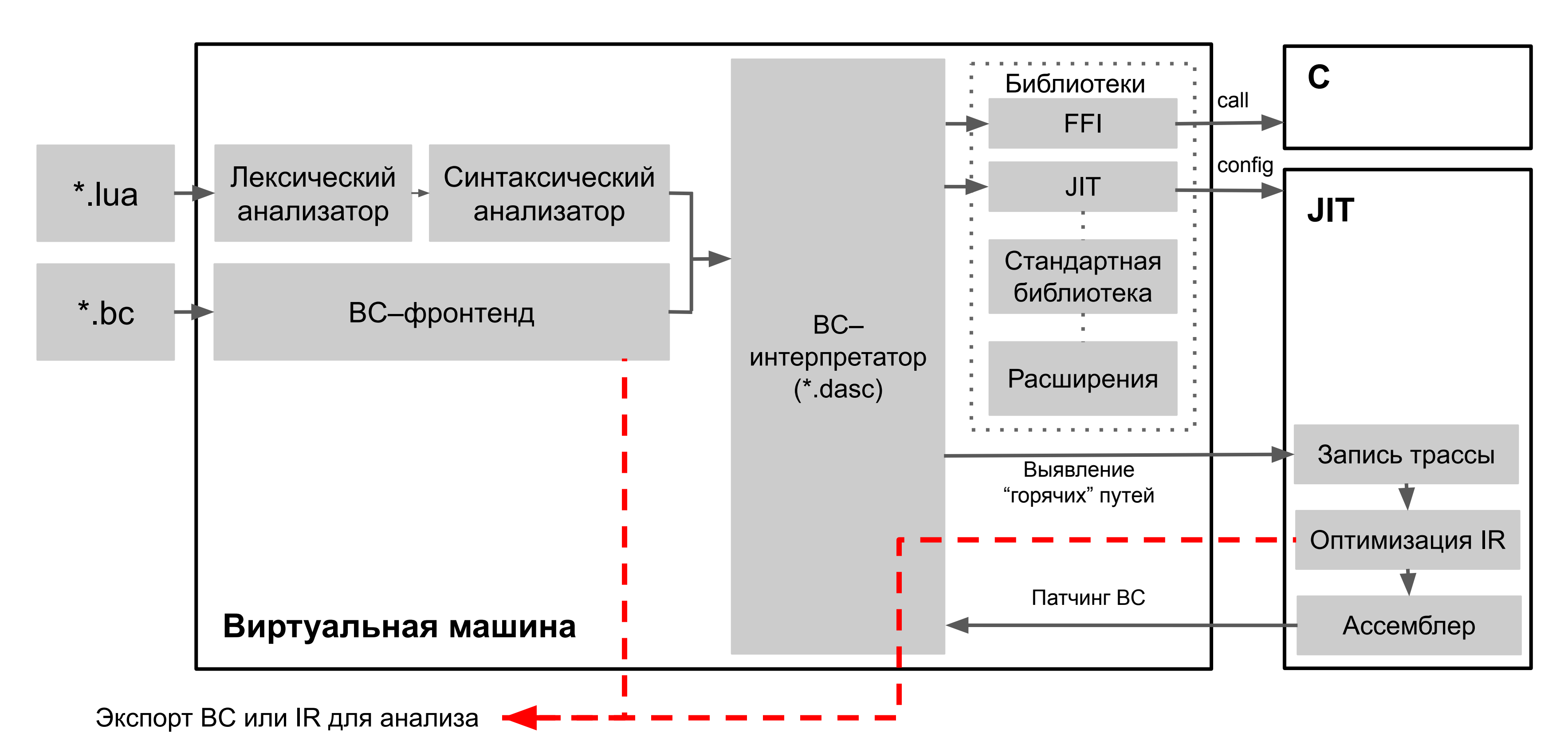
```
$ gcc pr106523.c -O1 -o pr106523

1, 6        # 64
7, 4        # 112
152, 19     # 196 != 4
```

Реализация для LuaJIT



Оптимизации на схеме LuaJIT



Управление JIT-компилятором

```
jit.on()      -- включает компиляцию
jit.off()     -- выключает компиляцию
jit.opt.*     -- контроль над оптимизациями


jit.opt.start(2)             -- выбрать уровень оптимизаций
jit.opt.start("-dce")        -- выключить оптимизацию dead code elimitation
jit.opt.start("hotloop=1")   -- агрессивно компилировать циклы
```

Что такое SMT-решатель?

“Задача выполнимости формул в теориях (англ. satisfiability modulo theories, SMT) — это задача разрешимости для логических формул с учётом лежащих в их основе теорий.” – Wikipedia

SMT-решатель: задача

SMT-решатель: решение

```
from z3 import *

circle, square, triangle = Ints('circle square triangle')

s = Solver()

s.add(circle + circle == 10)

s.add(circle * square + square == 12)

s.add(circle * square - triangle * circle == circle)

print(s.check(), s.model())


sat, [triangle = 1, square = 2, circle = 5]
```

Pro et Contra

- ✓ Математически строгий подход
- ✓ Фокусное тестирование оптимизаций компилятора
- ✗ Сложность в **аккуратной** трансляции семантики BC/IR в логические формулы

Выводы

- Фаззинг – это сложно!
- Фаззинг хорошо дополняет стандартные тесты
- Фаззинг позволяет заменить стандартные тесты
- Фаззинг это полностью автоматическое тестирование!

Сергей Бронников
Телеграм: @ligurio

Слайды и материалы:
brnkv.ru/hl2022

Голосуй за мой доклад!





Список найденных багов в LuaJIT

- "Assertion `ls->p < ls->pe' failed: lj_bcread.c:122: uint32_t bcread_byte(LexState *)" TNT#4824
- “Fix narrowing of unary minus.” TNT#6976
- “Assertion "return bytecode expected" failed when exit on error from a snapshot for stitched traces.” LUAJIT#913.

Секретные слайды

Пример проверки корректности оптимизации в BC/IR

Пример: программа на Lua

```lua
local function hl()
    local b
    for i = 1, 3 do
        b = 20          -- присвоение константы
    end
    return b
end
```

BC/IR без оптимизации: `luajit -jdump=-m -O0 -Ohotloop=1 example.lua`

BC/IR с оптимизациями: `luajit -jdump=-m -O3 -Ohotloop=1 example.lua`

Пример: BC/IR **без оптимизации**

```
---- TRACE 1 start example.lua:9
0006  KSHORT    0  20
0007  FORL      1 => 0006
---- TRACE 1 IR
0001      num SLOAD  #2    I
0002      num ADD    0001  +1
0003 >    num LE     0002  +3
0004      num CONV   +20   num.int
---- TRACE 1 stop -> loop
```

Пример: BC/IR **с оптимизацией**

```
---- TRACE 1 start example.lua:9
0006  KSHORT    0  20
0007  FORL      1 => 0006
---- TRACE 1 IR
0001    int SLOAD  #2    CI
0002  + int ADD    0001  +1
0003 >  int LE     0002  +3
0004 ------ LOOP ------------
0005  + int ADD    0002  +1
0006 >  int LE     0005  +3
0007    int PHI    0002  0005
---- TRACE 1 stop -> loop
```

Три правила преобразования кода в SMT–формулу



Пример: SMT-LIB без оптимизации

```
; объявляем переменные
(declare-const b1 Int)
(declare-const b2 Int)
; разворачиваем цикл (loop unroll)
(assert (= b1 20))    ; итерация 1
(assert (= b2 20))    ; итерация 2
```

Пример: SMT-LIB с оптимизацией

```
(declare-const b3 Int)

(assert (= b3 20))
```

Пример: проверяем эквивалентность

```
; проверяем равенство значений, возвращаемых из функции
(assert (= b3 b2))
(check-sat)
(get-model)
```

```
$ z3 z3_example.smt
sat
```

SAT